DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Tercer Trabajo Práctico

Junio de 2010 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Bianchi, Mariano	92/08	marianobianchi08@gmail.com
Brusco, Pablo	527/08	pablo.brusco@gmail.com
Di Pietro, Carlos Augusto Lyon	126/08	cdipietro@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Ejercicio 1

1.1. Introducción

El problema a resolver en el presente trabajo práctico consiste en dado un grafo simple, encontrar un *MAX-CLIQUE* para dicho grafo. Una clique es un subgrafo completo del grafo original. Un *MAX-CLIQUE* , es una clique tal que no exista otra que contenga más vértices.

Este problema es muy conocido. Además, no está computacionalmente resuelto y tiene infinidad de aplicaciones en distintos problemas de la vida real, lo que hace que sea un importante objeto de estudio. Algunas de sus aplicaciones más estudiadas provienen de áreas como bioinformática, transporte, diseño de tuberías, diseño de redes energéticas, procesamiento de imágenes, seguridad informática, electrónica, etc.

1.2. Algunas aplicaciones

1.2.1. Telefonía Móvil

Una aplicación posible podría darse por ejemplo, en el contexto de una compañía de telefonía móvil. Como bien se conoce, este tipo de empresa ofrece un plan llamado "plan empresas" para el cuál todos los teléfonos que se encuentren bajo este, tienen la posibilidad de comunicarse entre sí de forma libre.

Para la empresa, podría ser de interés conocer algún grupo de personas que estén comunicados todos entre sí para ofrecerles un "plan empresas" y así beneficiarlos dándole la oportunidad de aumentar el caudal de llamadas entre sí, por un precio más razonable.

Podemos pensar el modelo de la siguiente manera:

- Vértices: Son los celulares de los clientes de la empresa de telefonía.
- Ejes o aristas: Existe una arista entre dos vértices (o teléfonos) A y B si alguna vez se realizó una llamada entre ambos (A llamó a B o viceversa)[1].

Con este modelo, encontrar una clique de tamaño K significa encontrar un grupo de K celulares que se hayan comunicado todos entre sí alguna vez durante un período de tiempo predeterminado. Pasa lo mismo si se busca un *MAX-CLIQUE* . Esto sería equivalente a encontrar el mayor grupo de personas que estén comunicados todos entre sí[2].

De la misma forma, se puede pensar al revés, y quizás a la empresa le interese conocer grupos de celulares que estén todos comunicados entre sí, para NO ofrecerles el "plan empresas" ya que de esa manera ese grupo de personas podría eventualmente bajar el consumo de sus llamadas decrementando así las ganancias de la empresa de telefonía.

[1] Sería conveniente elegir un período de tiempo acotado para ver si se produjo dicha llamada o no y así poder armar el grafo.

[2] Una variante sería buscar el *MAX-CLIQUE* durante un año todos los días y quedarse con aquellos que se haya repetido la mayor cantidad de veces.

1.2.2. Planes Sociales

Imaginemos que el gobierno de la Nación quiere lanzar un nuevo plan social y quiere que la entrega de estos sea lo más equitativa posible entre la población. En este sentido, quiere entregar planes a la mayor cantidad de personas posibles de forma tal que ninguna persona que reciba el beneficio del plan esté relacionada directamente con otra que también lo reciba. Por relacionada directamente, se entiende que esas personas no tengan un parentesco directo que las una (madre,padre,hijo/a).

Podemos pensar el siguiente modelo de grafos:

- Vértices: Son las personas que son potencialmente beneficiarios del plan (i.e: mayores de 18 años que tengan hijos).
- Ejes o aristas: Existe un eje que une un par de vértices si existe un parentesco directo que una a las dos personas que representan esos nodos.

Entonces lo que se debería buscar en este modelo es el mayor conjunto de nodos independientes, es decir, un conjunto de nodos tales que para un par cualquiera de esos nodos no exista una arista que los una. Esto no es directamente transferible a un problema de *MAX-CLIQUE* , pero sí lo es si tomamos el complemento del grafo proveniente del modelo anteriormente descripto. Haciendo esto, sabemos que el grafo resultante tiene ejes donde antes no los había y le faltan aquellos que antes sí existían. Por lo que, si antes había un conjunto de nodos independientes, en el complemento en su lugar hay una clique. En consecuencia, ahora sí podemos ver que encontrar una *MAX-CLIQUE* en el complemento del grafo creado como antes se mencionó es igual a encontrar un grupo de personas no relacionadas entre sí para poder asignarles un plan social.

1.2.3. Buscador Web

Como bien sabemos, en un buscador web se realizan muchísimas búsquedas diarias. Para la empresa que mantiene un buscador web, puede ser de gran importancia saber cuál es el tema o palabra más buscado/a en algún período de tiempo en particular.

Podemos pensar por ejemplo el siguiente modelo para las búsquedas realizadas:

- Vértices: Representan una palabra o frase que haya sido buscada en el sitio web.
- Ejes o aristas: Las aristas aparecen entre dos nodos si entre esas frases hay alguna palabra en común (o un cierto porcentaje de palabras en común, por ejemplo, para evitar que dos frases estén relacionadas sólo por tener una preposición en común).

Para este modelo, encontrar una clique significa encontrar un conjunto de frases que (en cierto sentido) hacen referencia a una misma temática. Por lo tanto, encontrar una *MAX-CLIQUE* sería equivalente a conocer cuál es el tema (o palabra) más consultado en el buscador web.

2. Ejercicio 2

2.1. Introducción

En esta sección se presentara un algoritmo exacto para resolver el problema de encontrar la Clique Máxima en un grafo.

Aún no se conocen algoritmos buenos, es decir, polinomiales con respecto al tamaño de la entrada para resolver este problema. Así que nos consentraremos en realizar mejoras al algoritmo de fuerza bruta que considera todos los casos.

2.2. Explicación

Un algoritmo de fuerza bruta para resolver el problema de Max-Clique podría simplemente intentar formar el conjunto más grande de nodos, donde ese conjunto sea completo, intentando todas las posibilidades eligiendo todos los conjuntos de un cierto tamaño, luego intentar con un tamaño menor, etc. Probablemente la complejidad de un algoritmo de este estilo sea n^n donde n es la cantidad de nodos del grafo.

Una mejora que surge casi inmediatamente es utilizar la técnica de BackTracking, cuya función principal es intentar podar el arbol implicito de combinaciones posibles.

De todas maneras, implementar solo un BackTracking parece ser poco con respecto a las mejoras que se pueden lograr. A continuación, se explicara el algoritmo implementado con algunas mejoras, su pseudocodigo, y se verá cada una de las mejoras por separado.

Algoritmo Exacto(G: grafo)

```
1  CliqueMayor = vacia
2  componentesConexas = DetectarComponentesConexas(G)
3  Para cada componente en componentesConexas {
4      heap = crearHeap(G,componentes)
5      Mientras noVacio(heap) ∧ top(heap) ≥ tam(CliqueMayorActual){
6        v = top(heap)
7        Para tamCliqueABuscar desde grado(v)+1 hasta tam(CliqueMayorActual)+1{
8          vecinosFiltrados = filtrarVecinosMenores(v, tamCliqueABuscar-1)
9          Si tam(vecinosFiltrados) +1 ≥ tamCliqueABuscar
10           temp = BuscoCliqueDeTamañoK(tamCliqueABuscar, vecinosFiltrados)
11           Si tam(CliqueMayorActual) < tam(temp)
12             CliqueMayorActual = temp
13       }
14     }
```

Algoritmo 1: Pseudocódigo del algoritmo exacto

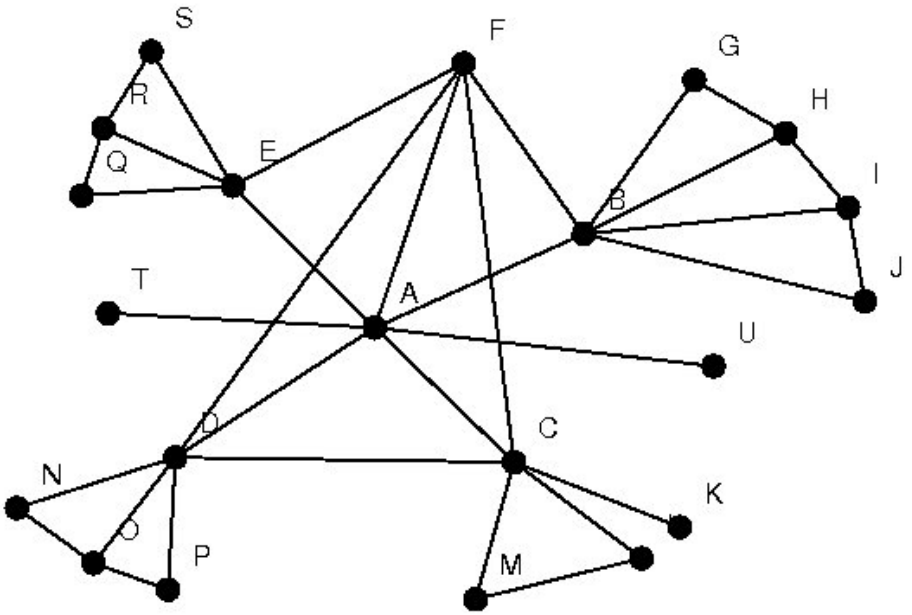
Primero, se detectan las componentes conexas del grafo, ya que buscar la max clique en todo el grafo es equivalente a quedarse con la máxima de las max cliques de cada componente conexa.

Luego, para cada componente, se crea un heap que contiene los nodos de la componente ordenados por mayor grado. Esto no parece tener mucha importancia, pero se aclarará a medida que se avanza con la explicación.

A partir de esta estructura, vamos obteniendo en cada iteración, el nodo v de mayor grado disponible en la componente que no hayamos analizado.

Una vez que tenemos el nodo v, lo que se hace es buscar la mayor clique en la cual está contenido. Para ello utilizamos la función BuscoCliqueDeTamañoK a la cual le indicamos el tamaño de la clique que queremos buscar (desde el grado de $v + 1$) y los vecinos de v que, dado su grado, tienen posibilidades de pertenecer a la clique.

Por ejemplo, veamos la siguiente figura:



Como se ve en el ejemplo, si buscamos cliques que contengan al vertice A (cuyo grado es 7), no tendria sentido buscarlas de cualquier tamaño, sino desde el grado del vertice más uno, es decir, cliques de tamaño 8 en este caso. Pero si vemos cuantos nodos vecinos de grado 7 tiene, notamos que no alcanzan para formar una clique de 8. Luego buscamos nodos vecinos de A de grado 6 o más y notamos que hay 3 (B, C, D) y A, por lo tanto tampoco alcanzan para una clique de tamaño 7. Luego los de grado 5 o más, encontramos 5 más A. En este caso, hay probabilidades de que estos 6 nodos puedan formar una clique de tamaño 6, entonces realizaremos una busqueda exaustiva utilizando backtracking, el cual en este caso dara negativo ya que no hay cliques de grado 6 en el grafo que contengan a A. Una vez más,

buscaremos una clique de tamaño 5, para lo cual solo miraremos nodos de grado 4 o más y realizaremos el backtracking sobre los nodos disponibles.

De todas maneras, el algoritmo intentará buscar cliques cada vez mas chicas hasta llegar a cliques de del tamaño de la más grande encontrada hasta el momento (sin incluir). Esto es así puesto que no tiene sentdo seguir buscando cliques más chicas que la mayor encontrada hasta el momento.

El heap de nodos ordenados por grados tiene gran importancia, ya que permite decidir rapidamente cuando terminar con la busqueda. Esto se realiza comprobando que el nodo siguiente del heap, es decir, el de mayor grado de la componente, supere o iguale al tamaño de la clique mas grande encontrada, ya que en caso contrario, no tendría sentido seguir buscando dentro de esa clique.

2.3. Resultados

A continuación se presentan los gráficos provenientes de las pruebas realizadas con archivos generados al azar. La forma en la que fueron generados estos archivos de entrada es explicada en la sección [8].

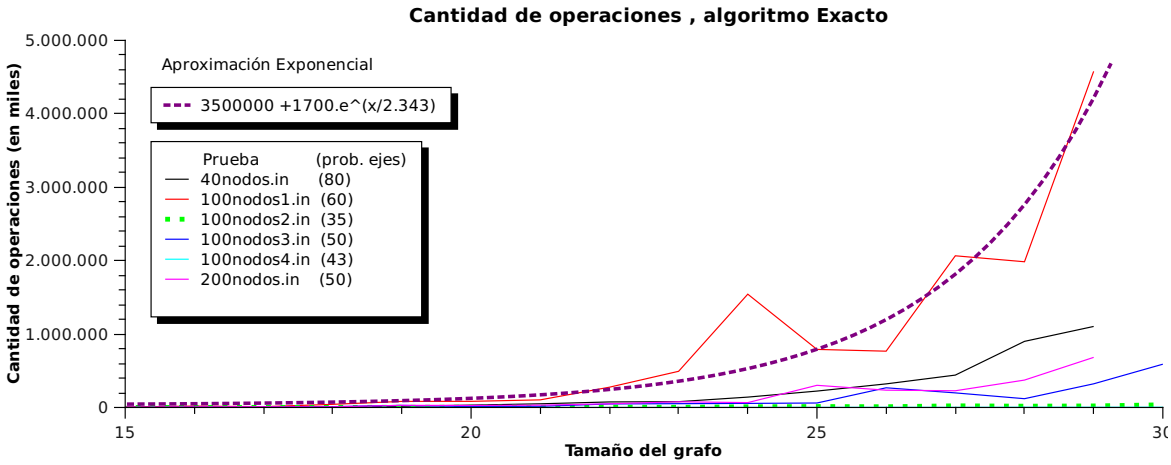


Figura 1: Muestra el comportamiento del algoritmo exacto, comparando el rendimiento del algoritmo para distintas cantidades de nodos contra cantidad de operaciones realizadas efectivamente. La probabilidad de aparición de ejes en estas muestras fueron las presentes en la tabla entre paréntesis. También, aparece una función exponencial generada con qti plot en base a los datos de la tabla que aproxima una de las mediciónes (100nodos1.in en este caso).

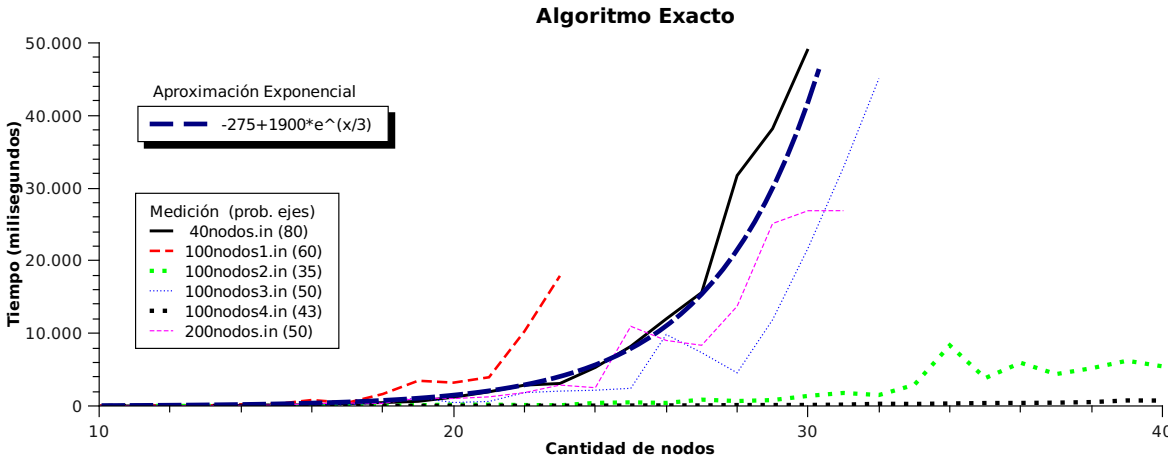


Figura 2: Muestra el comportamiento del algoritmo exacto, comparando el rendimiento del algoritmo para distintas cantidades de nodos contra el tiempo que demoro en devolver los resultados. La probabilidad de aparición de ejes en estas muestras fueron las presentes en la tabla entre paréntesis. También, aparece una función exponencial generada con qti plot en base a los datos de la tabla que aproxima una de las mediciónes (40nodos.in en este caso)

2.4. Debate y conclusiones

Viendo los gráficos podemos ver que para estos casos, el algoritmo se comporto como se esperaba (Exponencialmente) salvo algunos casos donde pareciera estar acotados por alguna función polinomica. Sin embargo en general esto no ocurre.

La única diferencia notable entre distintas pruebas fue que en los casos de prueba en los que había más ejes, es decir, en aquellos casos en que los grafos eran más densos, el algoritmo empeora, excepteptuando casos donde el grafo era muy denso (superior al 80 % de los ejes presentes) en cuyo caso funciona un poco mejor por lo explicado en la sección [2.2].

Podemos concluir entonces que suponer que el algoritmo exacto no es bueno, es decir, es exponencial, era correcto.

3. Ejercicio 3

3.1. Introducción

Para la resolución de este ejercicio se debía desarrollar e implementar una heurística constructiva para resolver el problema de encontrar un *MAX-CLIQUE* dado un grafo simple.

3.2. Explicación

En una primera aproximación al problema, se pensó un algoritmo bastante sencillo. La idea del mismo radicaba en ir tomando los nodos en orden de grados, es decir, comenzando con los de mayor grado hasta llegar a los de menor grado. De esta forma, uno puede pensar que al tomar primero los vértices de mayor grado, hay mas chances de encontrar una clique de mayor tamaño.

Esto es claramente una heurística válida que utiliza la técnica de algoritmo goloso. Pero es claro también que se pueden encontrar fácilmente ejemplos de grafos en los que dicho algoritmo funcione tan mal como uno quiera.

A fines de evitar en cierto grado muchos casos para los cuales este algoritmo funciona mal, se planteó uno nuevo que utiliza la misma idea pero que la misma no se realiza sobre todos los nodos del grafo sino que se hace sobre un subconjunto de los mismos. Para formar dicho conjunto, se implementó un algoritmo que revisa todas las combinaciones de 2 vértices distintos (siempre y cuándo haya 2 o más vertices en el grafo) que sean vecinos entre sí y se guarda en un conjunto de vértices aquellos que sean vecinos a ambos vértices y además se guardan los 2 vértices en cuestión. Esto se repite para cada posible combinación de vértices de a 2, guardando siempre el conjunto más grande que se haya encontrado completado de la forma antes mencionada.

Una vez encontrado este subgrafo, se procede a realizar el algoritmo goloso antes mencionado pero sobre dicho subgrafo, es decir, se busca el nodo con mayor grado en ese subgrafo y se coloca en un conjunto, el cuál será devuelto como clique al terminar el algoritmo. Luego, para el resto de los nodos del subgrafo, se va tomando de a uno a la vez en orden de mayor a menor grado (considerando sólo los adyacentes que pertenecen al subgrafo) y se verifica que sea adyacente a todos los que pertenecen a la clique. Si lo es, entonces se lo inserta en el conjunto sino se lo descarta. Finalmente, se prosigue con estos pasos hasta haber intentado con todos los nodos del subgrafo devolviendo entonces la clique encontrada.

A continuación se adjunta el pseudocódigo del algoritmo constructivo antes descripto y el de las funciones auxiliares pertinentes. En los mismos, utilizaremos a “n” como forma de expresar la cantidad de nodos pertenecientes al grafo.

cliqueConstructivo(G: grafo)

```
1  frontera = mayorFronteraEnComun(G) ;                    // O(n^3)
2  res = ∅;                                                // O(1)
3  mientras frontera ≠ ∅
4     v = elMasRelacionado(G,frontera);                    // O(n^2)
5     si esVecinoDeTodos(G,res,v);                         // O(n)
6        insertar v en res;                                // O(log(n))
7     fin si
8     eliminar v de frontera;                              // O(log(n))
9  fin mientras
10 si res == ∅;                                           // O(1)
11    insertar v_0 en res;                                 // O(1)
12 fin si
13 devolver res
```

Algoritmo 2: Pseudocódigo del algoritmo constructivo

mayorFronteraEnComun(G: grafo)

```
1  aux = ∅;                                                // O(1)
2  res = ∅;                                                // O(1)
3  paratodo u,v ∈ V_G tq (u,v) ∈ X_G
4     aux = (adyacentes(G,u) ∩ adyacentes(G,v)) ∪ u ∪ v;   // O(n)
5     si #aux > #res;                                      // O(1)
6        res = aux;                                        // O(1)
7     fin si
8  fin paratodo
9  devolver res
```

Algoritmo 3: Pseudocódigo de un algoritmo secundario al constructivo

El pseudocódigo de las funciones *elMasRelacionado, esVecinoDeTodos* y *adyacentes* no se detalla ya que no son algoritmos de gran complejidad. Igualmente, se explicarán brevemente a continuación.
En el caso de *elMasRelacionado*, recibe dos parámetros, un grafo y un conjunto de vértices. Esta función devuelve el vértice de mayor grado del grafo inducido por los vértices de ese conjunto. La función *esVecinoDeTodos* recibe un grafo, un vértice y un conjunto de vértices. Devuelve verdadero si y sólo si el vértice es adyacente a todos los vértices del conjunto. Por último, en el caso de la función *adyacentes*, recibe un grafo y un vértice como parámetros y devuelve el conjunto de todos los vértices adyacentes al pasado como parámetro.

3.3. Análisis de la complejidad del algoritmo

Para realizar el siguiente análisis de complejidad vamos a remitirnos al pseudocódigo de la función *AlgoritmoConstructivo* adjunto en la sección anterior. Cabe recordar que cada vez que se haga mención a "n" nos estaremos refiriendo a la cantidad de nodos del grafo al que se quiere analizar.
En la primer línea se realiza una llamada a la función *mayorFronteraEnComun* y una asignación. Estas 2 operaciones tienen una complejidad de $O(n^3)$. Más adelante se explicará el por qué de la misma. Luego, dentro del ciclo, la función que mayor complejidad temporal tiene es

elMasRelacionado y la misma es $O(n^2)$. Esta complejidad se debe a que para cada vértice del grafo inducido por el conjunto de vértices pasado como parámetro, se debe calcular cuántos vecinos tiene dentro de ese grafo, para lo que se debe recorrer todo el conjunto una vez por cada vértice. Como a lo sumo puede haber "n" vértices en ese conjunto, debo recorrerlo "n" veces por cada vértice. Entonces, su complejidad es $O(n^2)$.
Deberiamos ver ahora cuántas veces va a ejecutarse el ciclo. Este finalizará una vez que el conjunto llamado "frontera" quede vacío. Este arranca con el valor de la mayor frontera en común, por lo que a lo sumo puede ser de tamaño "n", es decir, todos los vértices pueden pertenecer a él en el peor caso. Podemos observar además, que en cada paso del ciclo, su tamaño disminuye en uno (línea 8, al eliminar un vértice del conjunto) , por lo que a lo sumo el ciclo se ejecutará "n" veces. Como la función más compleja dentro del ciclo era *elMasRelacionado* con una complejidad de $O(n^2)$ y la misma se ejecutará "n" veces, entonces la complejidad total temporal del ciclo es de $O(n^3)$. Por lo tanto, la complejidad del algoritmo constructivo es de $O(n^3)$.

Para completar este análisis, falta justificar por qué la función *mayorFronteraEnComun* tiene complejidad $O(n^3)$.
En el caso de la intersección, sólo hace falta recorrer una sola vez la lista de todos los vértices para saber cuáles son vecinos a ambos nodos a la vez, y la unión de dicha intersección con los vértices "u" y "v" tiene complejidad logarítmica, por lo que dicha instrucción tiene complejidad $O(n)$. Pero esta sentencia se ejecuta tantas veces como se ejecute el ciclo *for*, así que para saber la complejidad total, debemos conocer las veces que se ejecuta el ciclo: éste se ejecuta tantas veces como aristas haya. En el peor caso, si hay "n" nodos entonces puede haber hasta "n^2" aristas. Por lo que el ciclo se ejecutará a lo sumo n^2 veces[3]. Finalmente, la complejidad total de *mayorFronteraEnComun* es $O(n^3)$.

3.4. Resultados

A continuación se presentan los gráficos provenientes de las pruebas realizadas con archivos generados al azar. La forma en la que fueron generados estos archivos de entrada es explicada en la sección [8].

[3]La complejidad podría verse como $O(m)$ pero como se eligió una matriz de adyacencia para modelar el grafo, para recorrer todas las aristas es necesario pasar por cada valor de la matriz de adyacencia, por eso es que se toma como complejidad $O(n^2)$ y no $O(m)$

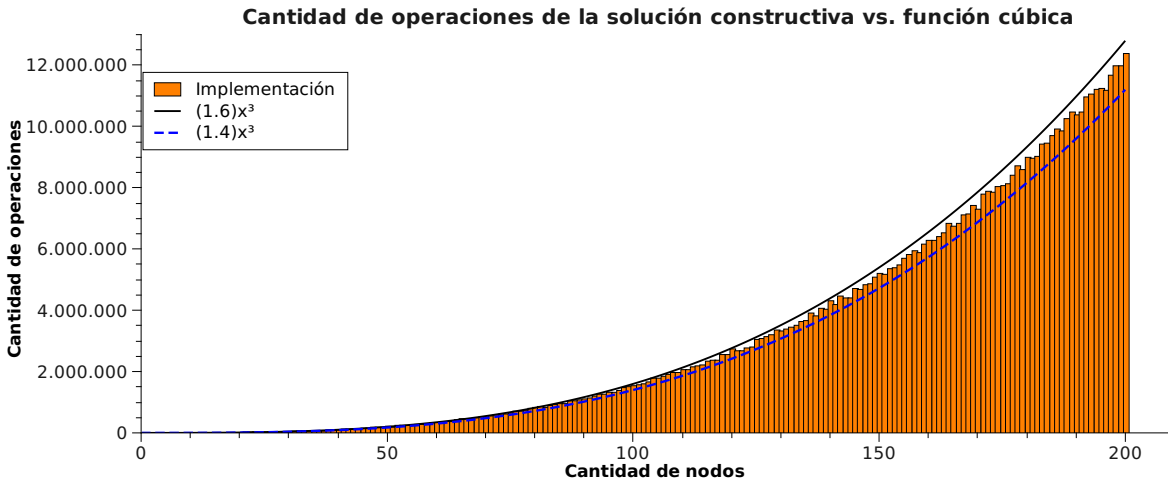


Figura 3: Muestra el comportamiento del algoritmo constructivo, comparando cantidad de nodos contra cantidad de operaciones. La probabilidad de aparición de ejes en esta muestra fue del 50 %

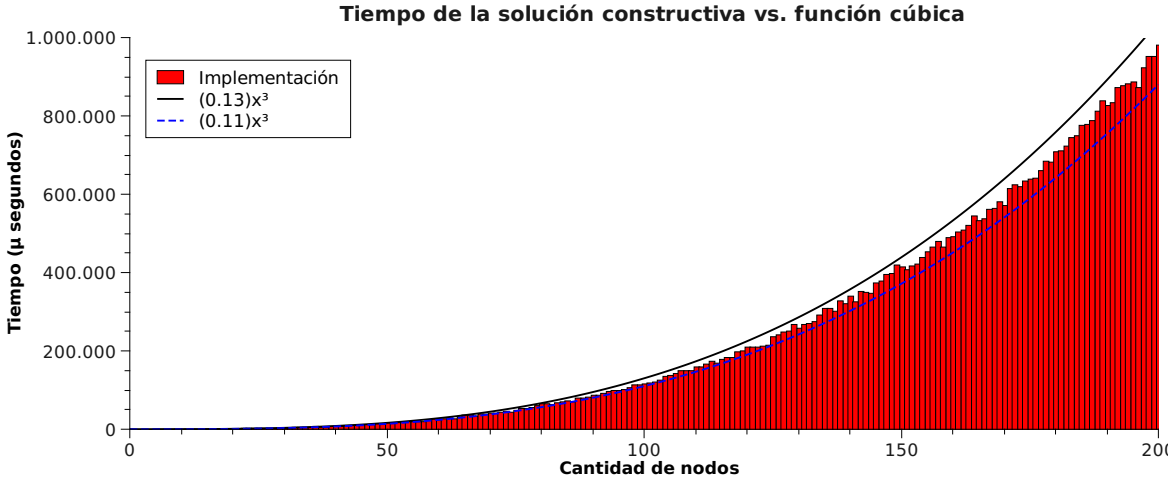


Figura 4: Muestra el comportamiento del algoritmo constructivo, comparando cantidad de nodos contra el tiempo medido en microsegundos. La probabilidad de aparición de ejes en esta muestra fue del 50 %

3.5. Debate y conclusiones

Si observamos los gráficos adjuntos en la sección anterior, podemos decir que para estos casos de pruebas el algoritmo constructivo pareciera haberse comportado como se esperaba. En ambos casos, los datos arrojados luego de realizar pruebas con el algoritmo pueden ser acotados tanto por debajo como por arriba por una función cúbica.

Para este análisis se realizaron varias pruebas, en las que se hizo variar tanto la cantidad de nodos pertenecientes a los grafos, como así también la cantidad de ejes. En todas ellas, pudo notarse una gran similitud en los datos arrojados, es decir, todas se comportaban como se esperaba y podían ser acotadas por arriba y por abajo por una función cúbica.
La única diferencia notable entre distintas pruebas fue que en los casos de prueba en los que había más ejes, es decir, en aquellos casos en que los grafos eran más densos, la constante que acompañaba a la función cúbica era un poco mayor. A pesar que esta diferencia existía, no era muy notable como para incluir gráficos que la mostraran.

Podemos concluir entonces que el análisis realizado en la sección [3.3] era correcto y que la complejidad de este algoritmo es de orden cúbico.

4. Ejercicio 4

4.1. Introducción

En este ejercicio, nos fue requerida una implementación de un algoritmo que utilice la técnica de búsqueda local para resolver el problema de *MAX-CLIQUE* .

4.2. Explicación

La idea general de la búsqueda local es partir de una solución parcial, ya sea una dada por algún algoritmo constructivo o alguna solución trivial y de allí realizar una búsqueda de soluciones vecinas que pueda mejorar la solución parcial encontrada antes.

Para comenzar con el algoritmo, se decidió que la solución de partida sea la que se obtiene a partir del algoritmo constructivo implementado como solución del ejercicio 3, que se detalla en la sección [3].
Luego, se definió qué iba a ser tomado como vecindad, es decir, qué conjunto de soluciones iban a ser tenidas en cuenta a la hora de realizar la búsqueda local. Para definir esta vecindad, hubo que tener en cuenta que la misma debía ser fácil de calcular. Además, el criterio de parada debía ser cuando se encontrara un máximo local.

Como la vecindad a definir debía ser lo suficientemente simple y debía mantener un espacio de búsqueda de soluciones relativamente pequeño, se optó por definir la siguiente vecindad:
Dada una solución parcial S, se saca un nodo v de dicha solución y se obtiene $S_1 = S \setminus \{v\}$, y se define un conjunto de candidatos a mejorar la solución parcial S. Sea C este conjunto de candidatos, los vértices pertenecientes a C son todos aquellos pertenecientes al grafo original que tienen al menos un vecino dentro de S_1. Una vez obtenido el conjunto de candidatos, se los va tomando en orden de grados (de mayor a menor) y se intenta insertarlos en S_1 siempre que esa inserción mantenga la propiedad de que S_1 sea una clique. Una vez hecho el intento con todos los vértices de C se verifica si se obtuvo o no una mejor solución que S, es decir, si $\#S_1 > \#S$. De ser positivo, esta solución S_1 es guardada.
Estos pasos se repiten para todos los vértices pertenecientes a S. Una vez finalizado esto, pueden suceder 2 cosas: que se haya encontrado una mejor solución que S o que no. Si se encontró una mejor solución, entonces se repiten todos los pasos antes mencionados pero para esta nueva solución parcial. Sino, significa que se encontró un máximo local, por lo que el algoritmo finaliza. Así como el orden en que se van tomando los vértices de C es de mayor a menor grado, en el caso de los vértices que se van descartando de S se hace en orden inverso, es decir, comenzando por los que tienen menor grado hacia los que tienen mayor grado.

Para graficar mejor este procedimiento, se detalla a continuación el pseudocódigo de las funciones que lo implementan. Recordar que dentro de los comentarios que contienen la complejidad, la letra "n" representa la cantidad de nodos de un grafo.

busquedaLocal(G: grafo)

```
1  termine = false;                                                    // O(1)
2  maxClique = cliqueConstructivo(G);                                  // O(n^3)
3  mientras termine == false hacer
4     cambio = false;                                                  // O(1)
5     <maxClique,cambio>= cambiarSiMaximiza(G,maxClique,cambio);
   // O(n^3 * log(n))
6     si cambio == false;                                              // O(1)
7        termine = true;                                               // O(1)
8     fin si
9  fin mientras
10 devolver maxClique
```

Algoritmo 4: Pseudocódigo del algoritmo de búsqueda local

cambiarSiMaximiza(G: grafo, maxClique: conjunto, cambio: bool)

```
1  copiaClique = maxClique;                                            // O(n)
2  cliqueDeMenorAMayor = deCliqueAMinHeap(copiaClique);
   // O(n * log(n))
3  mientras cliqueDeMenorAMayor no sea un Heap vacío
4     posibleMejora = copiaClique \ {tope(cliqueDeMenorAMayor)};
   // O(log(n))
5     candidatos = vecinosAMaxHeap(posibleMejora);          // O(n^2 * log(n))
6     mientras candidatos no sea un Heap vacío
7        si vecinoDeTodos(tope(candidatos),posibleMejora);             // O(n)
8           posibleMejora = posibleMejora ∪ tope(candidatos);   // O(log(n))
9        fin si
10       pop(candidatos);                                             // O(log(n))
11    fin mientras
12    si #posibleMejora > #maxClique
13       maxClique = posibleMejora;                                    // O(n)
14       cambio = true;                                                // O(1)
15    fin si
16    pop(cliqueDeMenorAMayor);                                        // O(log(n))
17 fin mientras
18 devolver <maxClique,cambio>
```

Algoritmo 5: Pseudocódigo del algoritmo cambiarSiMaximiza

4.3. Análisis de la complejidad del algoritmo

Para analizar la complejidad del algoritmo de búsqueda local basta con comprender cómo se comporta tanto el algoritmo principal, llamado "busquedaLocal", como el secundario llamado "cambiarSiMaximiza". El resto de los algoritmos puede obviarse ya que, o bien ya fueron analizados en la sección [3] como es el caso de "vecinoDeTodos", o bien tienen un comportamiento trivial y conocido, como es el caso de las funciones "deCliqueAMinHeap" y "vecinosAMaxHeap" que como sus nombres lo indican, son funciones que devuelven un min-heap y un max-heap respectivamente. Lo que si es interesante aclarar de estas funciones es qué elementos devuelven en cada heap. En el caso de "deCliqueAMinHeap" toma todos los vértices que pertenecen a la clique que se le pasa como parámetro y los heapifica según sus

grados (de menor a mayor). En cambio, la función "vecinosAMaxHeap" toma todos aquellos vértices que no pertenecen a la clique pero que tienen algún vecino dentro de ella y los inserta en un max-heap ordenándolos por grado.

En primer lugar vamos a analizar al segundo algoritmo aquí presentado, es decir, "cambiarSiMaximiza". Como se observa dentro del pseudocódigo, existen dos ciclos dentro del algoritmo, uno dentro de otro. Para que el ciclo más global finalice, debe ejecutarse tantas veces como elementos haya en "cliqueDeMenorAMayor" por lo que a lo sumo se ejecutará n veces. Pasa lo mismo con el ciclo anidado dentro de este. El mismo se ejecutará tantas veces como vecinos de la clique haya, es decir, a lo sumo n. Dentro de este segundo ciclo, la función más costosa es la que se encuentra en la guarda del if que tiene una complejidad de $O(n)$. Como este ciclo se ejecuta a lo sumo n veces, la complejidad del mismo es $O(n^2)$. Ahora bien, además del ciclo anidado, la otra función costosa dentro del ciclo mayor es "vecinosAMaxHeap" que cuesta $O(n^2 * log(n))$. Por lo tanto, como el ciclo mayor se ejecuta a lo sumo n veces, la complejidad total de este algoritmo es $O(n^3 * log(n))$.

Ahora bien, sólo faltaría analizar qué complejidad tiene la función principal. Pero si se observa bien, este algoritmo consta de un ciclo en el cuál existe una llamada a la función antes analizada. Por lo cuál sabiendo cuántas veces se ejecuta este ciclo, sabremos la complejidad total del algoritmo. Este ciclo finaliza cuando la variable "termine" valga $true$. La misma tomará este valor, cuando la variable "cambio" valga $false$ luego del llamado a "cambiarSiMaximiza". Por lo tanto, hay que volver al pseudocódigo de esta función para saber cómo se comporta allí esta variable.

La variable "cambio" sólo se modifica cuando existe una mejora en la clique mayor temporalmente hallada. Podemos ver entonces que el peor caso se da cuando la clique mayor temporal comienza con tamaño igual a 1 (respuesta obtenida con el algoritmo constructivo). Luego, dentro de cada llamada a la función "cambiarSiMaximiza", lo peor sería que sólo exista una mejora del tamaño en una unidad, es decir, en el primer llamado habrá un cambio de tamaño de 1 a 2, por lo que la variable "cambio" tomará el valor de verdad $true$. Cómo cada vez que se llama a esta función, esta mejora la respuesta temporal en una unidad, se puede observar que en el n-ésimo llamado el tamaño de la clique mayor temporal no aumentará, ya que se introdujeron todos los vértices existentes, por lo que en este llamado a la función secundaria, el algoritmo no entrará al if en el que se modifica a la variable "cambio", por lo que dicha variable quedará con valor $false$[4]. Por lo tanto, en el peor caso, en el n-ésimo llamado a esta función, la guarda del ciclo de la función principal se hace falsa. Entonces, como dicho ciclo se ejecutará a lo sumo n veces, y en el mismo hay un llamado a una función de complejidad $O(n^3 * log(n))$, la complejidad total del algoritmo principal es $O(n^4 * log(n))$.

4.4. Resultados

A continuación se presentan los gráficos provenientes de las pruebas realizadas con archivos generados al azar. La forma en la que fueron generados estos archivos de entrada es explicada en la sección [8].

[4]cabe notar que en cada llamado a la función "cambiarSiMaximiza" esta variable entra con valor $false$

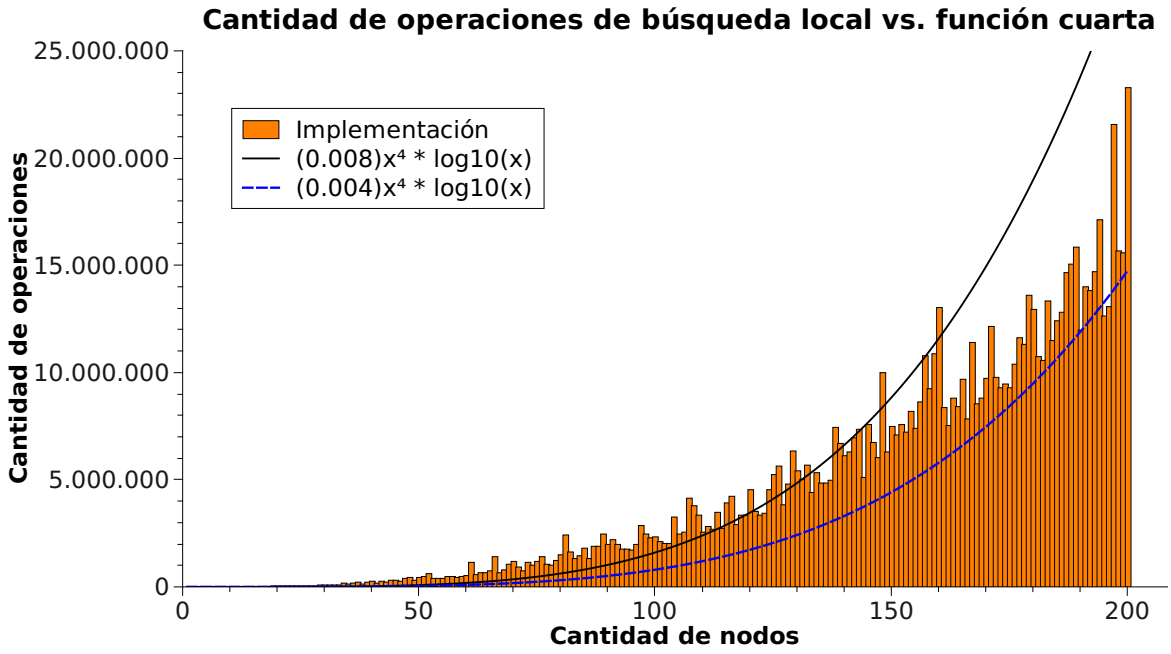


Figura 5: Muestra el comportamiento del algoritmo de búsqueda local, comparando cantidad de nodos contra cantidad de operaciones. La probabilidad de aparición de ejes en esta muestra fue del 50 %

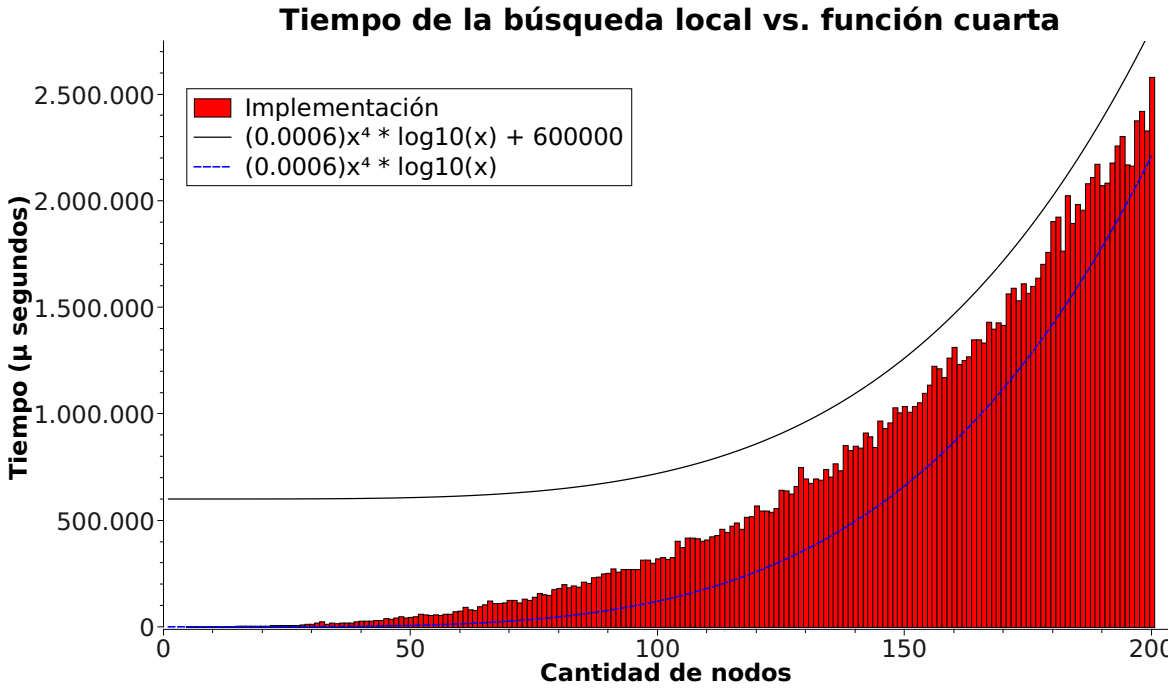


Figura 6: Muestra el comportamiento del algoritmo de búsqueda local, comparando cantidad de nodos contra el tiempo medido en microsegundos. La probabilidad de aparición de ejes en esta muestra fue del 50 %

4.5. Debate y conclusiones

Según los datos arrojados por los gráficos en la sección anterior, podemos ver que los datos arrojados por las distintas corridas del algoritmo de búsqueda local tienen la forma que se esperaba. En la sección [4.3], se explicó y analizó la complejidad de esta heurística, y se concluyó que la misma era $O(n^4 * log(n))$. Dentro de estos gráficos, se ve que en estos casos de prueba, la heurística parece haberse comportado como era esperado, y pudo ser acotada por arriba y abajo por una función cuarta multiplicada por un logaritmo.

Como se comentó en la sección de debate y conclusiones del ejercicio 3, a pesar de haberse tomado varias muestras en las que se variaba el número de ejes y de nodos, todos los resultados presentaban la misma forma, salvo alguna constante no muy pronunciada. Por ello no pareció adecuado incluir más gráficos, ya que en general y utilizando los generadores implementados por el grupo para este trabajo práctico, el comportamiento no varió de muestra en muestra.

Podemos concluir entonces que el algoritmo, tanto en la teoría como en la práctica para los casos de prueba realizados se comportó como se esperaba, arrojando una complejidad de $O(n^4 * log(n))$.

5. Ejercicio 5

5.1. Introducción

La búsqueda tabú es un algoritmo metaheurístico [5] basado en los algortimos heurísticos de búsqueda local, que aumenta su eficacia con respecto a este mediante el uso de estructuras de memorización. Esto le permite recodar características de las soluciones ya visitadas, para de esa forma evitar recorrerlas en el futuro, evitando así caer en ciclos dentro de la búsqueda.
Además, a diferencia de la heurísitica de búsqueda local, la metaheurística de búsqueda tabú cuenta con la posibilidad de admitir temporalmente soluciones peores con respecto a la mejor encontrada hasta el momento. Esta variación le brinda en algunos casos la posibilidad de continuar explorando en busca de la solución óptima en lugar de quedar reducida a una solución óptima local (máxima o mínima).

La búsqueda tabú utiliza un procedimiento similar al de búsqueda local o por vecindades moviéndose de forma iterativa desde una solución inicial S hacia una solución S' en $N(S)$ (la vecindad de S) que optimice el valor de la solución hasta satisfacer algún criterio de parada. Su principal diferencia radica en el uso de una la lista tabú, la cual es una estructura que le brinda al algoritmo una memoria de corto plazo y acotada [6], en donde puede alojar las soluciones que fueron visitadas en el pasado reciente.
Una variación que supone un uso más eficiente de esta esctructura consiste en la prohíbición de soluciones que tienen ciertos atributos o la prevención ciertos movimientos, puesto que el volumen de información a guardar en estos casos es significativamente menor que el necesario para guardar las soluciones obtenidas en las k iteraciones pasadas, siendo k el tamaño de la lista tabu.
De este modo, en cada iteración el algoritmo parte de una solución S hacia una solución S' que mejore el valor de la solución obtenida hasta el momento. Para ello, define una vecindad de S, $N(S)$, que representa el conjunto de soluciones alcanzables mediante transformaciones locales de S. Seguidamente, excluye de este conjunto a todas aquellas soluciones cuyos atributos esten registrados en la lista tabú o que sean generadas por transformaciones que estén en la lista tabú [7]. Finalmente, halla S' de entre las soluciones de $N(S)$, y el algoritmo marca a S como "tabú" incorporándola a la lista tabú continuando con su ejecución hasta alcanzar el criterio de parada.

En la presente parte de este trabajo se busca resolver el problema de *MAX-CLIQUE* mediante la implementación de un algoritmo metaheurístico basado en la técnica de búsqueda tabú.

[5] Las metaheurísticas son pues, heurísticas. No obstante, y generalmente la metaheurística consiste en estrategia de ataque del problema que cuenta con una heurística subordinada. Analizando su etimología, se obtiene que en griego meta: "más allá" y heurístico: "encontrar"; lo cual evidencia que la metaheurística supone un nivel superior en cuanto a la heurística.

[6] Generalmente el valor que acota el tamaño de la lista tabú es arbitrario pero dependiente del valor del tamaño de la entrada.

[7] Cabe aclarar que aún cuando sólo un atributo es marcado como tabú, esto por lo general resulta en que más de una solución sea marcada como tabú, y en consecuencia sea temporalmente ignorada.

5.2. Explicación

El concepto principal de la metaheurística desarrollada consiste en realizar repetidas veces una búsquedas tabú, partiendo cada vez de una solución inicial distinta. La motivación detrás de esta idea es la de, mediante esta técnica de diversificación, poder alcanzar espacios de búsqueda que de otro modo no sería explorados por el algoritmo, logrando así aumentar el número de soluciones potenciales cuyo valor [8] sea mayor que el de la mejor solución obtenida hasta el momento.

Para poder comprender de manera más cabal tanto la idea como el funcionamiento del algoritmo que implementa la metaheurística de búsqueda tabú se dividirá la explicación en dos partes. En primer lugar, se analizará el algoritmo encargado de diversificar las soluciones iniciales y realizar las sucesivas búsqueda tabú, el cual se halla implementado en la función *cliqueTabu*. En segunda instancia, se procederá a elucidar el algoritmo de búsqueda tabú propiamente dicho, el cual se encuentra implementado en la función *busquedaTabu*.

5.2.1. Primera Parte

Dando comienzo a la explicación de la función *cliqueTabu*, se presenta a continuación el pseudocódigo de la misma, y seguidamente se expone con mayor detalle el funcionamiento del algoritmo.

CliqueTabu(G: Grafo)

```
1  var res : Conj(ℤ) ;                                   // O(1)
2  var cliqueAux : Conj(ℤ) ;                             // O(1)
3  var n : ℤ;                                            // O(1)

4  n ← cant_nodos(G) ;                                   // O(1)
5  res ← cliqueConstructivo(G) ;                         // O(?)
6  cliqueAux ← res ;                                     // O(n)

7  Para 1, ... , n/2 { ;                                // O(?)

8     cliqueAux ← busquedaTabu(G, cliqueAux) ;           // O(?)

9     Si (tam(cliqueAux) > tam(res)) ;                   // O(1)
10       res ← cliqueAux ;                               // O(n)

11    cliqueAux ← ∅ ;                                    // O(1)
12    cliqueAux ← diversificar(G) ;                      // O(n)
13 }

14 return res ;                                          // O(1)
```

Algoritmo 6: Pseudocódigo de la función cliqueTabu

Tal como puede observarse del pseudocódigo anterior, el algoritmo *cliqueTabu* toma como solución de partida la solución obtenida mediante la heurística constructiva desarrolada en el

[8] Téngase presente que el problema que se busca resolver es el de *MAX-CLIQUE* , y que por ende se entiende por posibles soluciones a los conjuntos de nodos que forman cliques, y por valor de una solución a la cantidad de nodos que la conforma. Precisamente, lo que se busca con esta metaheurística es encontrar la solución de mayor valor

Ejercicio 3 del presente trabajo. A partir de allí, el algoritmo comienza un ciclo que repite *n/2* veces, siendo n la cantidad de nodos del grafo analizado. En cada ciclo, se realiza una búsqueda tabú sobre esa solución inicial , y si encuentra una clique de tamaño mayor que la registrada hasta ese momento la guarda. Seguidamente, se procede a buscar una nueva solución inicial por medio de la función *diversificar* y se continua con la ejecución del ciclo. Finalmente, una vez que el ciclo concluye, el algorimo devuelve el cojunto de nodos que conforman la mayor clique que fue capaz de encontrar.

En este punto, y antes de continuar con la explicación de *busquedaTabu*, cabe hacer un pequeño paréntesis para comentar brevemente el funcionamiento de la función *diversificar*. El algoritmo implementado en dicha función concretamente lo que hace es brindar de modo aleatorio una solución trivial al problema. En otras palabras, devuelve un conjunto de nodos que sean trivialmente una clique; esto es o bien un único nodo, o bien un par de nodos adyacentes.
Para generar esta solución, el algortimo elige al azar un nodo v cualquiera del grafo, y seguidamente elige otro nodo u al azar de entre los vecinos de v. Finalmente, devuelve un conjunto con el par de nodos v,u en caso de que el $gr(v) > 0$, o devuelve un conjunto con v en caso contrario.

Finalmente, una observación para cerrar la exposición del algoritmo implementado en la función *cliqueTabu*. Nótese que este algoritmo en su primera iteración realiza la búsqueda tabú partiendo de la solución brindada por la heurística constructiva. Esto, a priori, supone una ventaja para la metaheurística ya que en los casos en que la búsqueda contructiva se acerca "de forma aceptable" a la solución óptima, la búsqueda tabú parte de una solución bastante encaminada con lo cual se vuelve esperable que encuentre la mejor solución.
Por el contrario, si la solución de la heurística constructiva corresponde a un caso patológico y alejado de la solución óptima, esto no necesariamente supone un problema para la metaheurística aquí expuesta, ya que en cada iteración dentro de la función *cliqueTabu* el algoritmo diversificará la búsqueda generando nuevas soluciones iniciales triviales a partir de las cuales realizar nuevamente la búsqueda tabú con intención de obtener resultados más favorables.

5.2.2. Segunda Parte

A continuación, se procederá a explicar el funcionamiento del algoritmo de búsqueda tabú. Para ello, y al igual que en el caso anterior, se incorpora al pie el pseudocódigo de la función *busquedaTabu* para acompañar la explicación.

busquedaTabu(G: Grafo, res: Conj($\mathbb{Z}$))

```
1  var agregados : bool ;                                          // O(1)
2  var n : ℤ;                                                      // O(1)
3  var v : ℤ;                                                      // O(1)
4  var u : ℤ;                                                      // O(1)
5  var stop : ℤ;                                                   // O(1)
6  var desmejore : ℤ;                                              // O(1)
7  var vecindad : Heap(ℤ) ;                                        // O(1)
8  var cliqueTemp : Conj(ℤ) ;                                      // O(1)
9  var TabuAgregados : Vector(ℤ)) ;                                // O(1)
10 var TabuEliminados : Vector(ℤ)) ;                               // O(1)

11 n ← cant_nodos(G) ;                                             // O(1)
12 stop ← 10 ∗ n ;                                                 // O(1)
13 desmejore ← 0 ;                                                 // O(1)
14 agregue ← false ;                                               // O(1)
15 cliqueTemp ← res ;                                              // O(n)

16 Para nro_it entre [0...stop) { ;                                // O(?)
17   Si ((desmejore == n/4) or (cliqueTemp == ∅)) ;                // O(1)
18     break ;                                                     // O(1)

19   v ← nodoConMenorGrado(G, nro_it, TabuAgregados, cliqueTemp) ;
     // O(?)
20   Si (v ≠ −1) { ;                                               // O(1)
21     borrar(cliqueTemp, v) ;                                     // O(log(n))
22     TabuEliminados[v] ← nro_it+n/2 ;                            // O(1)

23     vaciar(vecindad) ;                                          // O(n)
24     vecindad ← definirVecindad(G, nro_it, TabuEliminados, cliqueTemp)
     ;                                                             // O(?)

25     agregue ← false ;                                           // O(1)
26     Mientras (vecindad ≠ ∅)) { ;                                // O(n ∗ log(n))
27       u ← tope(vecindad) ;                                      // O(1)
28       pop(vecindad) ;                                           // O(1)
29       Si (vecinoDeTodos(u, cliqueTemp)) { ;                     // O(n)
30         agregue ← true ;                                        // O(1)
31         agregar(cliqueTemp, u) ;                                // O(log(n))
32         TabuEliminados[v] ← nro_it+n/2 ;                        // O(1)
33       }
34     }

35     Si (agregue == false) ;                                     // O(1)
36       desmejore++ ;                                             // O(1)

37     Sino si (tam(cliqueTemp) > tam(res)) { ;                    // O(1)
38       desmejore ← 0 ;                                           // O(1)
39       res ← cliqueTemp ;                                        // O(n)
40     }
41   }
42 }

43 return res ;                                                    // O(1)
```

Algoritmo 7: Pseudocódigo de la función busqudaTabu

En base al pseudocódigo anterior se observa que la función *busquedaTabu* recibe como parámetros el grafo sobre el cual se busca determinar la *MAX-CLIQUE* y un conjunto de nodos, *res*, que forman una clique dentro del mismo grafo. Al momento del inicio del algoritmo de búsquda tabú, este conjunto representa la mejor solución hallada en ese punto de la ejecución. Conforme el algoritmo avance, en *res* se irán guardando las cliques del grafo que tengan mayor cantidad de nodos que la clique guardada en res hasta ese momento; o eventualmente, en caso de que el algorimo no consiga hayar una clique mas grande, no se modificará en absoluto.

Como primer paso, el algoritmo declara varias variables e inicializa dos de ellas (*stop* y *desmejore*) que serviran como criterio de parada para la búsqueda [9]. Seguidamente copia el conjunto *res* recibido como parámetro en la variable *cliqueTemp*. Esta variable contendrá las sucesivas soluciónes que el algortimo vaya transformando y que registrará en *res* en caso de que alguna de todas estas soluciones resultantes tenga un valor mayor que la solución allí guadada al tiempo de ejecución. Luego, el algoritmo ingresa al ciclo en el que hará efectiva la búsqueda tabú y del cual saldra únicamente cuando se halla cumplido alguno de los criterios de parada.

Una vez dentro del ciclo, la primera acción realizada por el algoritmo es evaluar si se cumple alguna de las condiciones de parada. De ser así la busquedaTabu se interrumpe y se vuelve a la función cliqueTabu para diversificar la solución e iniciar una nueva búsqueda tabú. En el caso contrario, se continúa con la ejecución del ciclo.

Seguidamente, se ejecuta la función *nodoConMenorGrado* y guarda el resultado en la variable *v*. Esta función devuelve el nodo de menor grado de la *cliqueTemp* que no se encuentre inhabilitado en el vector *TabuAgregados*; y en caso de que todos los nodos de *cliqueTemp* estén marcados como prohibidos en el vector *TabuAgregados*, entonces *nodoConMenorGrado* devuelve -1.
La idea de elegir este nodo *v* tiene por fin quitarlo de la *cliqueTemp* para modificar esa solución de forma tal que luego sea posible agregar nuevos nodos. El hecho de que sea el de menor grado busca preservar los nodos de mayor grado dentro de la solución con vista a maximizarla. Asimismo, el hecho de que el nodo elegido no se encuentre entre nodos prohibidos en el vector *TabuAgregados* pretende no desandar el camino ya recorrido al eliminar alguno de los nodo que fue agregado en alguna de las iteraciones recientes. Precisamente, aqui puede observarse como el algoritmo hace uso de la memorización para no volver sobre sus pasos al cotejar el número de iteración actual(*nro_it*) con los números de iteración a partir de los cuales cada nodo puede ser volver a ser utilizado contenidos en el vector *TabuAgregados*, pudiendo estableces así si un nodo está o no vetado.
Acto seguido, si la función *nodoConMenorGrado* devolvió un nodo inválido (v == −1) el algortimo no hace nada y vuelve a iterar con la intención de que algún nodo prohibido en TabuAgregados haya dejado de serlo. Sino, si *nodoConMenorGrado* devuelve un nodo válido, el algoritmo elimina *v* de la *cliqueTemp* y lo prohibe por *n/2* iteraciones, asignando para ello, *nro_it* aumentado en *n/2* en la posición *v* del vector *tabuEliminados*. Esto permite ir teniendo registro de los nodos que fueron eliminados y que no podrán volver a ser agregados hasta que el *nro_it* del ciclo no alcance los valores de esos nodos dentro del vector *tabuEliminados*.

[9] Estos criterios será explicados más adelante, estando ya más adentrados en el comportamiento del algoritmo.

Habiendo eliminado ya uno de los nodos de *cliqueTemp*, el paso siguiente del algoritmo consiste en definir el conjunto de nodos que intentará agregar a *cliqueTemp*. En este sentido, se setea el flag *agregue* como falso, se vacia el heap *vecindad*, y luego se ejecuta la función *definirVecindad* almacenando su resultado en *vecindad*. La función *definirVecindad* busca el conjunto de todos los nodos que estan fuera de *cliqueTemp* pero que tengan por vecino al menos a algún nodo de esa clique. Luego, elimina de ese conjunto a todos aquellos nodos que esten proscriptos por el vector *TabuEliminados*, y ubica los nodos restantes en un maxheap ordenándolos por grado. Este heap es el que devuelve la función.
Nuevamente aquí, el hecho de que los nodos que se busca agregar a *cliqueTemp* esten ordenados en un heap tiene por objetivo poder ir agregando aquellos de mayor grado primero en pos de maximizarla. Además, otra vez puede apreciarse como el algoritmo recurre al uso de la memorización para elegir de entre los posibles nodos a agregar únicamente aquellos cuyo valor en el vector *TabuEliminados* es menor o igual al de la iteración actual. De este modo evita, al menos por un tiempo, no volver a evaluar soluciones ya visitadas.

Una vez establecidad la vecindad de nodos que se busca incorporar a *cliqueTemp* se procede a ir a tomando el nodo ubicado en el tope de la vecindad e incorporalo únicamente si éste es vecino de todos los nodos en *cliqueTemp*, repitiendo el proceso hasta que el heap quede vacio. Al mismo tiempo, a cada nodo agregado *cliqueTemp* se le asigna el valor *nro_it + n/2* en su correspondiente posición dentro del vector *TabuAgregados* para que de ese modo quede prohibido eliminarlo durante las siguientes *n/2* iteraciones [10]. En cualquier caso, si se hubiese agregado como mínimo un nodo, la variable *agregue* pasa a tener el valor true.

A continuación, se evalua si durante la iteración actual no se agregaron nuevos nodos a *cliqueTemp* (agregue == false), en cuyo caso representa una desmejora, y se procede a incrementar el contador *desmejore*. Dicho contador, busca registrar la cantidad de veces que *cliqueTemp* disminuyó su tamaño con respecto a su valor durante la iteración previa. Es así que en el caso en que se quitan un nodo sin agregarse nuevos este contador aumenta, mientras que en el caso en que se encuentra una solución mejor respecto a la contenida en *res* el contardor se resetea. Esta variable permite establecer un criterio de parada mediante la fijación de una cota para los casos en que el algoritmo entra en una curva de desmejoramiento de la clique. En la presente implementación se estableción que el algoritmo pudiese desmejorar a lo sumo *n/4* veces. En caso de alcanzarse la cota, la *busquedaTabu* se interrumpe y se procede a diversificar la búsqueda volviendo a la función *cliqueTabu*.
Otra situación que puede darse es que el conjunto contenido en *cliqueTemp* quede vacío antes de que la variable *desmejore* alcance la cota [11]. Dicho caso también constituye el segundo criterio de parada del algoritmo, puesto que resulta ineficiente continuar la optimización sobre una solución vacía. Por el contrario, resulta más eficiente realizar detener la ejecución, diversificar y volver a correr la *busquedaTabu*.

[10] Una observación interesante en este punto respecto a la implementación es el hecho de que ambos vectores, *TabuAgregados* y *TabuEliminados* funcionan de forma análoga a una cola *FIFO*, puesto que los primeros nodos en prohibirse son los primeros en pasar a estar disponible una vez pasados sus tiempos de prohibición.

[11] A modo de ejemplo puede pensarse en el caso de un grafo grande y denso en el que la solución constructiva es una clique pequeña alejada del óptimo. Partiendo de esa solución, si el algortimo entrase rápidamente en una curva de demejoramiento antes de que la solución pudiese crecer lo suficiente *cliqueTemp* quedaría vacío mucho antes de que *desmejore* alcance la cota establecida por el criterio de parada.

En cambio, si se agregaron nodos a *cliqueTemp* y el tamaño resultante es mayor que la solución guardada en *res*, se resetea el contador *desmejore* y se guarda en *res* el conjunto de nodos contenido en *cliqueTemp*.

Finalmente, queda esclarecer el tercer y último criterio de parada. Este no es ni más ni menos que la cota que detiene al ciclo **Para** del pseudocódigo expuesto anteriormente. La misma es un valor almacenado en la variable *stop* que es inicializada al comienzo del algoritmo con el valor *10*n*. Este criterio de parada resulta necesario ya que no hay certeza de que alguno de los otros dos ocurra alguna vez.

Una vez fuera del ciclo, por cualquiera de los tres criterio de parada, la función *busquedaTabu* devuelve el mayor conjuto de nodos del grafo que conforman una clique, el cual se encuentra almacenado en *res*, y finaliza.

5.3. Análisis de la complejidad del algoritmo

En esta sección del informe, se expone el análisis de complejidad de la metaheurística de búsquda tabú implementada para resolver el problema de *MAX-CLIQUE* .

Analizado el psegudocódigo de la función *cliqueTabu*, presente en la primera parte de la explicación del algoritmo, se pude apreciar que la complejidad de la misma está dada principalmente por la complejidad de la función *cliqueConstructivo* más la complejidad del ciclo **Para**.
La complejidad de *cliqueConstructivo*, tal como se explicó en la parte de este informe correspondiente al Ejercicio 3, es $O(n^3 * log(n))$. Por su parte, se puede decir sin necesidad de conocerla, que la instrución más costosa del ciclo **Para** en cuanto a complejidad es la llamada a la función *busquedaTabu*. Dado que el ciclo itera *n/2* veces, la complejidad del mismo debe ser del orden de n*complejidad(*busquedaTabu*).
En consecuencia, se puede afirmar que la complejidad de la función *cliqueTabu* es:

$$O(n^3 * log(n) + n * complejidad(busquedaTabu))$$

Luego, solo resta averiguar la complejidad de la función *busquedaTabu*. Para ello, es necesario previamente, conocer las complejidades de las funciones *nodoConMenorGrado* y *definirVecindad* ambas ejecutadas desde un ciclo **Para** dentro de la función *busquedaTabu*.
A continuación, se incluyen los pseudocódigos de dichas funciónes para poder comprender y calcular el valor de sus respectivas complejidades:

nodoConMenorGrado(G: Grafo, nro_it: $\mathbb{Z}$, TabuAgregados: Vector($\mathbb{Z}$), cliqueTemp: Conj($\mathbb{Z}$))

```
1  var res : ℤ;                                          // O(1)
2  var v : ℤ;                                            // O(1)
3  var aux : Conj(ℤ) ;                                   // O(1)

4  Para v en cliqueTemp { ;                              // O(n)
5     Si (TabuAgregados[v] ≤ nro_it) ;                   // O(1)
6        agregar(aux, v) ;                               // O(log(n))
7  }

8  Si (aux == ∅) ;                                       // O(1)
9     res ← -1 ;                                         // O(1)

10 Sino {
11    res ← dameUno(cliqueTemp) ;                        // O(1)
12    Para v en aux { ;                                  // O(n)
13       Si (grado(G,v) > nro_it) ;                      // O(1)
14          res ← v ;                                    // O(1)
15    }
16 }

17 retunr res ;                                          // O(1)
```

Algoritmo 8: Pseudocódigo de la función nodoConMenorGrado

Como puede verse en el pseudocódigo anterior, la complejidad de la función *nodoConMenorGrado* esta dada por la suma de las complejidades de ambos ciclos **Para**. El primer ciclo, que filtra aquellos nodos que no están prohibidos en el vector *TabuAgregados*, puede llegar a realizar una inserción en un conjunto por cada una de las n veces que se ejecuta. El segundo puede tener en el peor caso un costo de complejidad lineal en la cantidad de nodos.
En consecuencia, la complejidad esta función es $O(O(n*log(n))+O(n))$; pero esto, analizado asintóticamente es igual a $O(n*log)$ [12].

definirVecindad(G: Grafo, nro_it: ℤ, TabuEliminados: Vector(ℤ), cliqueTemp: Conj(ℤ))

[12]La complejidad de la función *grado* que figura en el pseudocódigo y que devuelve el grado de un nodo dentro del grafo que lo contien es $O(1)$ puesto a que está implementada sobre un arreglo.
La complejidad de la función *dameUno* se considera $O(1)$ porque en la implementación se toma el primer elemento del conjuto.

```
1  var v : ℤ;                                                    // O(1)
2  var u : ℤ;                                                    // O(1)
3  var res : Heap(ℤ) ;                                           // O(1)
4  var aux : Conj(ℤ) ;                                           // O(1)

5  Para v en cliqueTemp { ;                                      // O(n)
6     Para u entre [0..cantNodos(G)) { ;                         // O(n)
7        Si(sonAdyacentes(G,v,u) and (cuenta(cliqueTemp,u) == 0) ;
   // O(log(n))
8           agregar(aux, nodo) ;                                // O(log(n))
9  }

10 Para v en aux { ;                                             // O(n)
11    Si(TabuEliminados[v] > nro_it) ;                          // O(1)
12       agregar(aux, nodo) ;                                   // O(log(n))
13 }

14 res ← ponerEnHeap(aux) ;                                      // O(n * log(n))

15 retunr res ;                                                  // O(1)
```

Algoritmo 9: Pseudocódigo de la función nodoConMenorGrado

Al igual que en el caso anterior, la complejidad de la función *definirVecindad* está dada nuevamente por la complejidad del primer ciclo que ejecuta. Este ciclo **Para** tiene anidado otro ciclo del mismo tipo, por lo cual en el peor caso realiza n^2 iteraciones. Además, cada iteración tiene un costo de complejidad $O(2 * log(n))$ dado que la función *cuenta* tiene costo $O(log(n))$ por la búsqueda en un conjunto de enteros, y la inserción de un elemento en un conjunto de enteros tiene el mismo costo. Por lo tanto la complejidad de este ciclo no es otra más que $O(n^2 * O(2 * log(n)))$, lo cual equivale asintóticamente a $O(n^2 * log(n))$
Realizando un análisis similar, se obtiene que la complejidad del segundo ciclo *Para* dentro de la función *definirVecindad* tiene una complejidad $O(n * log(n))$, mientras que la función *ponerEnHeap* es de costo $O(n * log(n))$ debido a la inserción lineal de enteros en un heap.
Puesto que las complejidades de ambos ciclos junto con la de la función *ponerEnHeap* se suman, resulta entonces que la complejidad de la función *definirVecindad* es $O(n^2 * log(n) + 2 * n * log(n))$, o lo que es lo mismo asintóticamente, $O(n^2 * log(n))$.

En resumidas cuentas, hasta el momento se sabe que la complejidad de *cliqueTabu* es $O(n^3 * log(n) + n * complejidad(busquedaTabu))$ y que la complejidad de *busquedaTabu* depende de las complejidades de las funciones *nodoConMenorGrado* y *definirVecindad*, las cuales por lo desarrollado previamente se sabe que son $O(n * log(n))$ y $O(n^2 * log(n))$ respectivamente.
Resta entonces dilucidar la complejidad de la función *busquedaTabu*. Analizando el pseudocódigo de esta función se puede ver que básicamente el algortimo es un gran ciclo *Para* en el cual la mayoria de las operaciones tiene costo computacional constante o logarítmico en la cantidad de nodos, a excepción de las llamadas a las funciones *nodoConMenorGrado* y *definirVecindad* y la ejecución de un ciclo *Mientras*. Las dos primeras complejidades ya fueron explicadas anteriormente, por lo que son conocidas. Por otra parte, se puede afirmar que la complejidad del ciclo *Mientras* es $O(n * log(n))$ ya que revisa los elementos de un heap, que en el peor caso puede contener a todos los nodos del grafo, y eventualmente los inserta en un

conjunto.
Por consiguiente, la función *busquedaTabu* ejecuta en cada iteración de su ciclo *Para* instrucciones de un costo equivalente a $O(n^2 * log(n)) + 2^* O(n * log(n))$. Como en el caso más desfavorable el ciclo puede llegar a iterar *10*n* veces, resulta entonces que la complejidad de *busquedaTabu* es $O(n * (O(n^2 * log(n)) + 2 * O(n * log(n)))$; lo cual en el análisis asintótico recae en $O(n^3 * log(n))$.

Habiendo, averiguado la complejidad de la función *busquedaTabu*, resulta posible entonces concluir que la complejidad del la métaheurística de búsqueda tabu que resuelve el problema de *MAX-CLIQUE* , implementada en la funcion *cliqueTabu* es:

$$O(n^3 * log(n) + n^4 * log(n))$$

5.4. Resultados

A contnuación se presentan los gráficos elaborados en base a los datos recolectados por las sucesivas pruebas y experimentos con la metaheurística implementada sobre los archivos de prueba utilizados en los ejercicios anteriores:

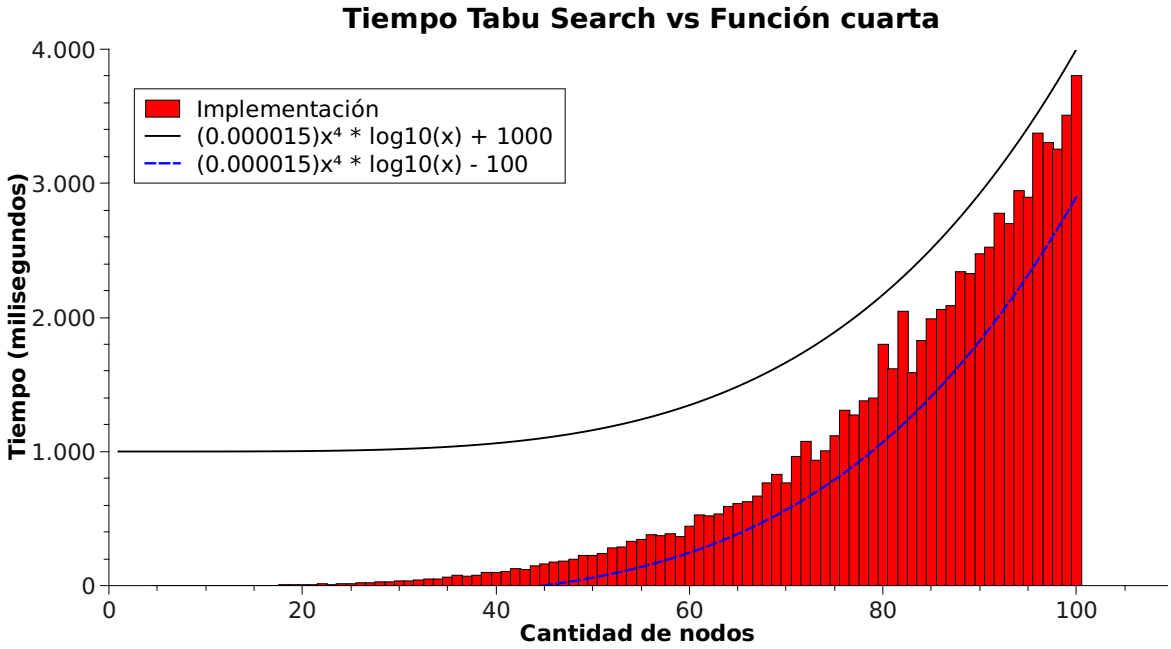


Figura 7:

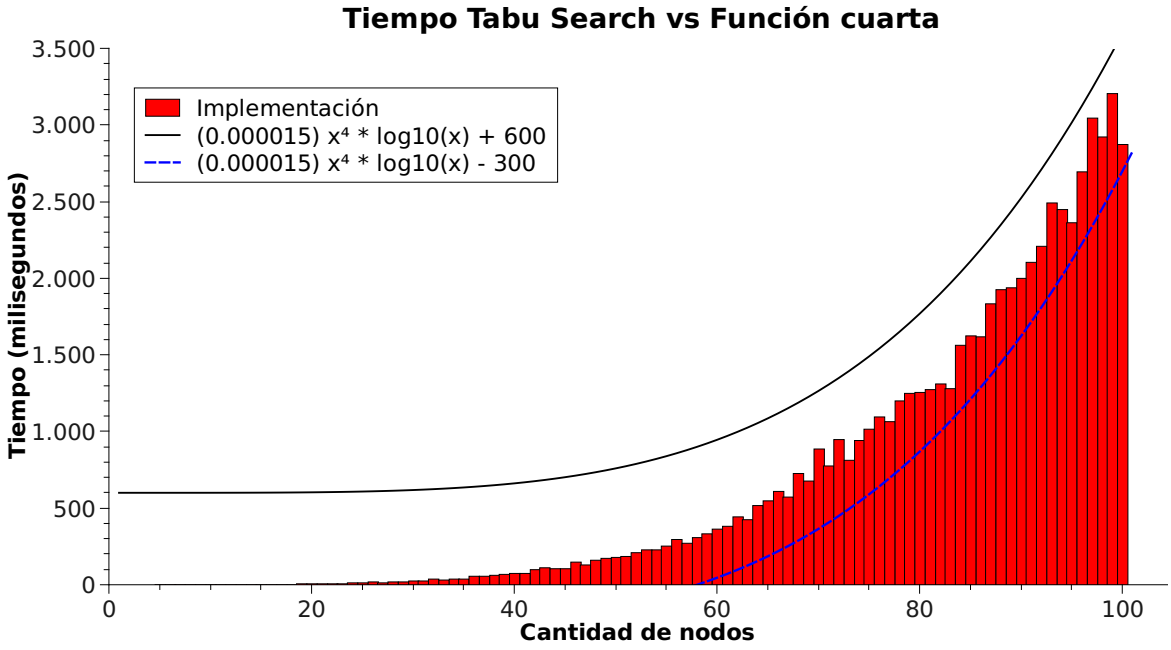


Figura 8:

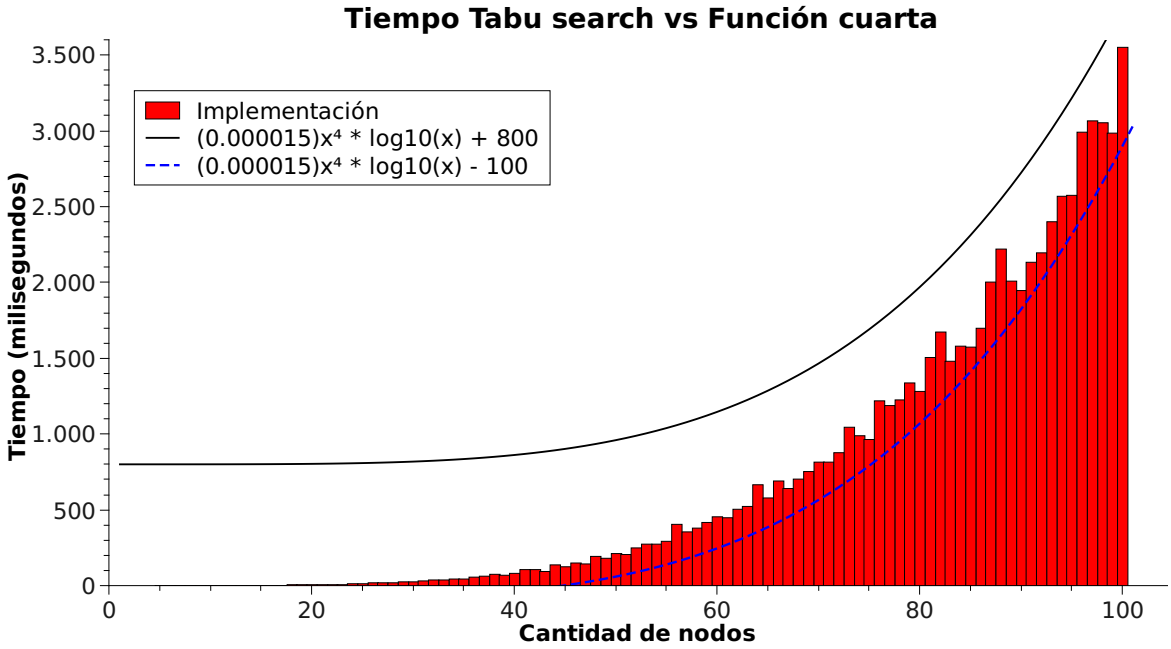


Figura 9:

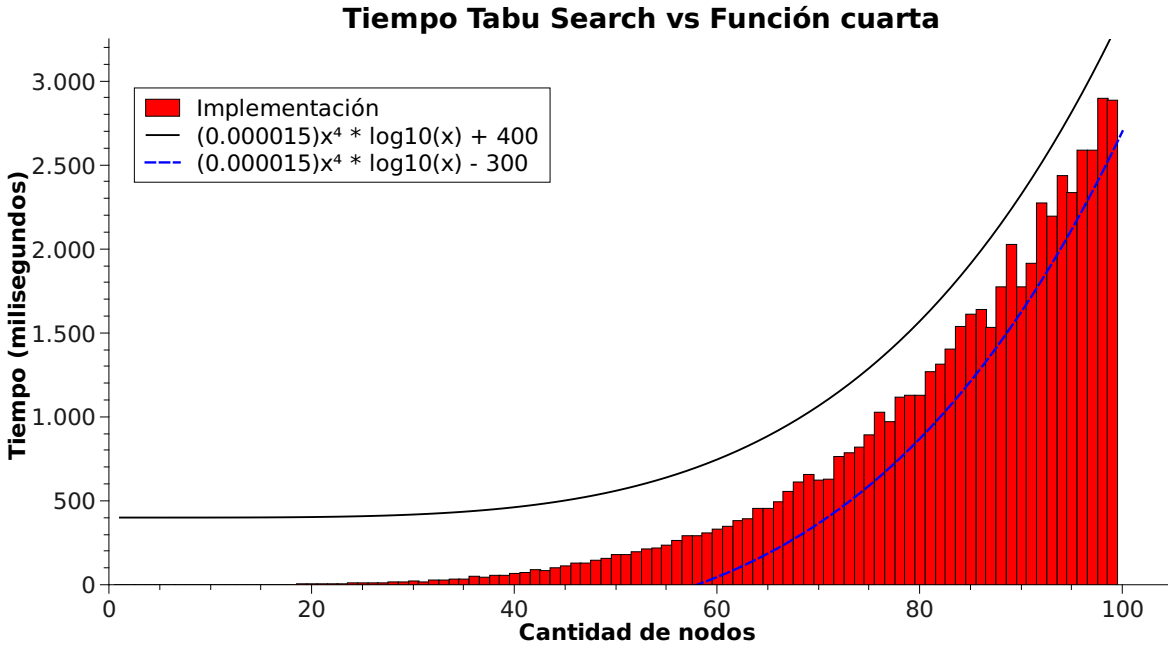


Figura 10:

5.5. Debate

El principal aspecto esperable del comportamiento de la metaheurística de búsqueda tabú es que requiera un mayor período de tiempo para resolver los mismos casos de prueba que las demás heurísitcas implementadas en este trabajo. La motivación de esta presunción surge de tener en cuenta que si bien la metaheurística demora un tiempo significaticamente menor en relación el algoritmo exacto para obtener una solucion [13], ésta se idea con el propósito de mejores que soluciones que las demás heuristicas mediante el uso de diversas técnicas y métodos de exploración del grafo. Por tanto cabe presuponer que dicha intensificación en la búsqueda va acompañada de un consecuente aumento de cómputo que finalmente se traduce en un incremento del tiempo de ejecución del algoritmo.

El segundo aspecto esperable, y que prácticamente se desprende del primero es que las soluciónes de la metaheurísticas sean mejores que las de los restantes algoritmos. Esto es, que para una misma instancia, la clique encontrada mediante búsqueda tabú sea mayor en tamaño que las cliques encontradas mediante las heurísticas constructiva y local. Esta hipótesis se sostiene en el hecho suponer que el mayor nivel de computo que posee la metaheurísticas por sobre las otras heurísticas es empleado de forma eficiente y efectiva, y que por ello conlleva a una solución de igual o mayor calidad.

Una tercera hipótesis, es la de esperar que el tiempo de ejecución de la metaheurística y las soluciones que genera generalemnte empeoren en el caso en que la misma no parte de la solución constructiva. Esta teoría se apoya en la concepción de que la metaheurística es una estrategia de ataque a un problema que se monta sobre una heurística subordinada. En otras palabras, es esperable debido a que al empezar a explorar a partir de una respuesta obtenida por otra heurística se tiene una solución base y no se empieza desde cero.

Por último, se espera que el algoritmo se comporta acorde al análisis teórico de complejidad realizado previamente en este informe. Debiera ser posible observar en la práctica que los tiempos de ejecución son acotables por alguna función del estilo $f(x) = c.x^4 * log(x) + d$, con c y d contantes enteras.

En resumen, las hipótesis que uno podría formular en base al desarrollo y el estudio realizado sobre la metaheurística son las siguientes:

1. La metaheurística de búsquda tabú demora más que las demás heurísticas de este trabajo en obtener una solución.
2. La metaheurística de búsquda tabú obtiene una mejor solución que las demás heurísticas en este trabajo.
3. La metaheurística de búsquda tabú demora más y obtiene peores soluciones cuando no parte de una solución generada por la heurística constructiva.
4. Los tiempo de ejecución de la metaheurística de búsquda tabú se comportan acorde al análisis teórico de complejidad.

[13] Es lógico que así sea pues esa es básicamente la motivación para desarollar una heurística

5.6. Conclusiones

Una de las primeras cosas que se puede inferir de los gráficos presentados en la sección anterior es que en las pruebas realizadas la metaheurística se comportó de la manera prevista, ya que en todos ellos se pudo acotar los tiempos tanto inferior como superiormente por una función $f(x) = c.x^4 * log(x) + d$, con c y d contantes enteras. En consecuencia, se puede afirmar que la hipótesis 4 se vió corroborada en la práctica.

Otro de los hechos observables a partir de los gráficos es que los tiempos de ejecución de la metaheurística son menores respecto a los de la heurística constructiva y significativamente respecto a los de la heurística de de búsqueda local. Esto en principio llevaría a desetimar la hipótesis 1. No obstante, las comparaciones de tiempos de ejecución presentadas en la sección Comparaciones corroboran la hipótesis de manera tajante. Esto lleva a inferir que los resultados obtenidos en esta parte la experimentación se debe a alguna caracteristica particular de los grafos generados, y no al comportamiento del algoritmo en general. Posiblemente esa característica este vinculada con el grado de densidad del grafo. Sin embargo, no se cuenta con evidencia suficiente como para afirmar esta última suposición.

En lo concerniente a la hipótesis 3, las pruebas realizadas evidenciaron que al correr la metaheurística sin partir de una solución constructiva, los tiempos de ejecución se incrementaron, y las soluciones en muchos casos fueron peores. En particular, para la quinta instancia del archivo *"TP3.in*, la metaheurística sin partir de la solución constructiva no es capaz de encontrar la *MAX-CLIQUE* , mientras que la heurística constructiva sí. Esto puede deberse a que el comportamiento de la heurística se basa en quitar un nodo primero y luego intentar agregar nuevos, en lugar de probar el comportamiento inverso; es decir, agregar primero y quitar después.

Finalmente, en cuanto a la hipótesis 4, la comparación de los archivos *".out"* presentes en la carpeta *out* dentro del cd que se presenta junto con este trabajo demuestra que en líneas generales los resultados obtenidos por la metaheuristica superan a los de la heurística constructiva, pero no a los de la búsqueda local. Asimismo, las comparaciones presentadas en la sección Comparaciones tienden a corroborar estos hechos. Por estos motivos, la hipótesis no puede ser confirmada.
Esta inferioridad no esperada en el nivel de las soluciones puede deberse a la forma en que se generaron los archivos de prueba con los que se testeo el algoritmo, los cuales puede estar explotando alguna debilidad del algoritmo en relación a algún tipo de caso no contemplado durante su desarrollo. Posiblemente, las soluciones no alcanzables en esos casos se vean imposibilitadas por alguna caracteristica de implementación de la lista tabú como pueden ser su tamaño y/o el tiempo de permanencia.

6. Detalles de implementación

Dentro de la carpeta que contiene todos los archivos de este trabajo práctico, puede encontrar un script de python llamado *tp.py* que tiene por fin facilitar el uso del todos los programa aquí desarrollados.
Este script puede ser ejecutado desde una consola, siendo necesario para ello situarse en el directorio en que se encuentra el script e introducir luego la instrucción `python tp.py`. Una vez ejecutado el script, se presenta en pantalla un menú con las siguientes opciones:

1 Recompilar el programa todos los programas [14].

2 Recompilar un programa en particular

3 Ejecutar todos los programas

4 Ejecutar un programa en particular

5 Salir

Una vez compilado/s el/los programa/s que se desea/n ejecutar, eligiendo una de las opciones 2 o 3 respectivamente, se mostrará en pantalla un segundo menú que lista todos los archivos *".in"* ubicados en la carpeta *./in*, permitiendo elegir el aquel que se desee ejecutar o en su defecto, todos ellos.
A medida que los archivos *".in"* vayan siendo procesados se mostrará en pantalla el tiempo promedio que tardó cada algoritmo en resolver todas las instancias contenidas en el archivo. Asimismo, una vez finalizada la ejecución de el/los programas se mostrará un mensaje en pantalla avisando la finalización de el/los mismo/s y se deberá presionar enter para volver al menu inicial.
Las respuestas a las instancias de cada archivo *".in"* estarán en un archivo *".out"* homónimo dentro del directorio *./out/Nro_de_Ejercicio*

Se recomienda que cualquier archivo *".in"* que se desee resolver mediante alguno de los programas sea colocado en la carpeta *./in* para poder ser elegido desde el menu que muestra el script [15].
De cualquier forma, todos los programa también pueden ser compilados por consola con el comando `make` y pueden ser ejecutados con el comando `./ej#nro_de_ejericio` recibiendo como parámetro las rutas de los archivos de entrada *".in"* a procesar. Pueden recibir tantos archivos como se desee, pero en caso de no recibir ninguno, se procesará el archivo *./in/Tp3.in*. En todos los casos, los archivos correspondientes archivos *".out"* serán guardados en las correspondientes carpetas *./out/Nro_de_Ejercicio*.

Además, en la carpeta del cd entregado junto con este trabajo se halla una carpeta con el código fuente del generado de grafos utilizado, y una carpeta correspondiente a cada ejercicio. En el interior de cada una de esta últimas se encuentran los archivos fuentes en lenguaje C++ del programa desarrollado, los cuales están cometados para simplificar la comprensión

[14] Todos los programas son compilados para sistemas operativos GNU-Linux

[15] En el directorio *./in* hay un archivo ejecutable llamado *generador* que genera archivos con varias instancias de grafos. El mismo puede ejecutarse desde consola por medio del comando `./generador`

de cada algortimo. Asimismo hay directorios varios con las variantes del codigo utilizado para realizar las mediciones tanto de cantidad de operaciones como de tiempo.

7. Comparaciones

7.1. Benchmarks

A modo de ampliar la cantidad de resultados anteriormente presentados y con intensión de verificar el comportamiento de las heurísticas implementadas a través de test más rigurosos y un poco más universales, nos vimos en la búsqueda de casos de test conocidos o un poco más estandarizados. En esta búsqueda, se dimos con una página en la web, en la cuál hay varios casos de prueba, no sólo para *MAX-CLIQUE* sino también para coloreo, cubrimiento de vértices, entre otros. La página en cuestión es: http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm. En ella no sólo están los casos de prueba utilizados en esta sección para comparar las heurísticas, sino que también figura una explicación de cómo los mismos fueron creados.

Cant. nodos	*MAX-CLIQUE*	Constructivo	Búsqueda local	Tabú Search
450	30	24(14s)	25(15s)	24(114s)
595	35	26(36s)	27(38s)	27(317s)
760	40	31(82s)	33(90s)	31(755s)
945	45	33(165s)	37(179s)	34(27.5min)
1150	50	40(315s)	40(324s)	40(54min)
1272	53	37(435s)	40(455s)	38(79min)

7.2. Un caso particular

En la búsqueda de ampliar las comparaciones entre los algoritmos, nos vimos en el trabajo de buscar algún ejemplo de grafo, si es que había, en el cuál se vieran plasmadas las mejoras que se van realizando ejercicio a ejercicio. Por esta razón, se realizó un análisis más detallado del comportamiento de cada algoritmo para ver si se podía cumplir con este objetivo.

Luego de realizar varios bosquejos, dimos con el siguiente grafo:

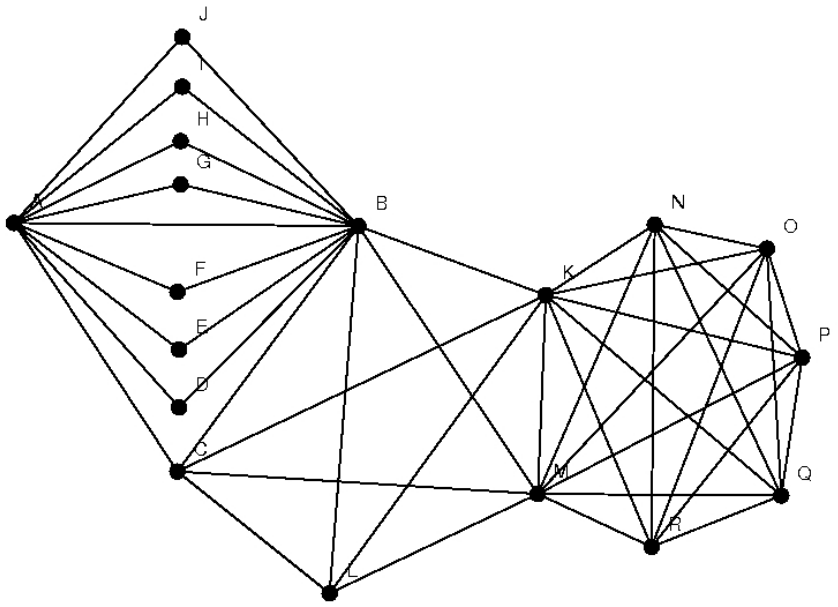


Figura 11: Grafo en el que se observan distintos resultados entre los distintos algoritmos. Hay una cantidad total de 18 nodos.

Algoritmo	***MAX-CLIQUE* encontrada**
Exacto	7
Constructivo	3
Búsqueda local	5
Búsqueda tabú	7

Cuadro 1: *MAX-CLIQUE* encontrada por cada algoritmo para el grafo de la Figura 11.

Cabe aclarar que el *MAX-CLIQUE* dentro de este grafo particular es de tamaño 7. Si se hace una corrida de máquina para cada algoritmo implementado en este trabajo práctico sobre este grafo, se observan los siguientes resultados:

Como se puede ver en esta tabla, existe una mejora en el resultado arrojado por cada heurística. Esto se debe principalmente a las técnicas algorítmicas utilizadas en cada ejercicio, y además a que tanto la heurística de búsqueda local implementada, como la de búsqueda tabú, parten de una solución inicial, que es la arrojada por el algoritmo constructivo. Por lo tanto, partiendo de ese detalle, sabemos que siempre va a valer que:

$$\#CliqueConstructiva \leq \#CliqueBusqLocal \leq \#CliqueExacta \tag{1}$$

y

$$\#CliqueConstructiva \leq \#CliqueBusqTabu \leq \#CliqueExacta \tag{2}$$

Por lo dicho anteriormente, jamás se va a encontrar un ejemplo en el cuál el resultado arrojado por la heurística constructiva sea mayor que el arrojado por cualquiera de las otras heurísticas.
El ejemplo de la Figura 11 es un claro ejemplo en el que las desigualdades de las ecuaciones cambian de menor o igual, a estrictamente menor (salvo el caso de la clique exacta y la tabú).

8. Anexos

Para los casos de prueba, se utlizaron grafos generados aleatoriamente de la siguiente manera:

- se ingresa cantidad de nodos.
- se ingresa una probabilidad de aparición de ejes.
- se genera una matriz de adyacencia en donde cada conexión tendra una probabilidad de aparecer igual a la ingresada.

De esta manera, no conocemos las cliques generadas, pero es una manera de generar casos aleatorios simples.